Noname manuscript No.
(will be inserted by the editor)

slim ABS

the date of receipt and acceptance should be inserted later

1 The language slim ABS

1.1 *Syntax*

Figure 1 displays `slim ABS` syntax, where an overlined element corresponds to any finite sequence of such element. The elements of the sequence are separated by commas. For example $\overline{T}$ means a (possibly empty) sequence $T_1, \cdots, T_n$. When we write $\overline{T\ x\ ;}$ we mean a sequence $T_1\ x_1\ ;\ \cdots\ ;\ T_n\ x_n\ ;$ when the sequence is not empty; we mean the empty sequence otherwise.

A program P is a list of method declarations followed by a *main function* $\{\,\overline{F\ x\ ;}\ s\,\}$. A type T is the name of either a primitive type `Int` or an object type `Object`. A type F can be a type T or a *future type* `Fut<`T`>`.

A statement s may be either one of the standard operations of an imperative language or one of the operations for scheduling. This operation is `await` x`?` (the other one is `get`, see below), which suspends method's execution until the argument x, is resolved. This means that `await` requires the value of x to be resolved before resuming method's execution.

Expressions z may have side effects (may change the state of the system) and include object creation `new Object`, *asynchronous* method call e`!m(`e`)`. Asynchronous method invocations do not suspend the execution of the caller. Expressions z also include the operation e`.get` that suspends method's execution until the value of e is computed. The type of e is a future type that is associated with a method invocation. The difference between `await` x`?` and e`.get` is that the former releases the object's lock when the value of x is still unavailable; the latter does not release the object's lock (thus being the potential cause of a deadlock).

A *pure* expression e is either a value, or a variable x, or the reserved identifier `this`. Values include the `null` object, and primitive type values (integers).

Address(es) of author(s) should be given

1.2 *Semantics*

`slim ABS` semantics is defined as a transition relation between *configurations*, noted *cn* and defined in Figure 2. Configurations are sets of elements – therefore we identify configurations that are equal up-to associativity and commutativity – and are denoted by the juxtaposition of the elements *cn cn*; the empty configuration is denoted by ε. The transition relation uses two infinite sets of names: *object names*, ranged over by o, o', $\cdots$, and *future names*, ranged over by f, f', $\cdots$. The function fresh( ) returns either a fresh object name or a fresh future name; the context will disambiguate between the twos.

Runtime values are either values v in Figure 1 or object and future names or an undefined value, which is denoted by $\perp$.

With an abuse of notation, we range over runtime values with v, v', $\cdots$. We finally use l, possibly indexed, to range over maps from local variables to runtime values. The map l also binds the special name destiny to a future value.

The elements of configurations are

– *objects* $ob(o,p,q)$ where o is an object name; p is either idle, representing inactivity, or is the *active process* $\{l \mid s\}$, where l returns the values of local identifiers and s is the statement to evaluate; q is a set of processes to evaluate.

P	$::= \overline{M}\ \{\overline{F\ x\ ;}\ s\}$	program
T	$::=$ `Int` $\mid$ `Object`	type
F	$::=$ `T` $\mid$ `Fut<`T`>`	future type
M	$::= T\ \mathtt{m}(T\ x)\{\overline{F\ x\ ;}\ s\}$	method definition
s	$::=$ `skip` $\mid x = z \mid$ `if` $e\ \{s\}$ `else` $\{s\} \mid$ `return` $e \mid s\ ;\ s \mid$ `await` $e?$	statement
z	$::= e \mid e!\mathtt{m}(e) \mid$ `new Object` $\mid e$`.get`	expression with side effects
e	$::= v \mid x \mid$ `this` $\mid$ *arithmetic-and-bool-exp*	expression
v	$::=$ `null` $\mid$ *primitive values*	value

Fig. 1 The language `slim ABS`

$$cn ::= \epsilon \mid \mathit{fut}(f, \mathit{val}) \mid \mathit{ob}(o, p, q) \mid \mathit{invoc}(o, f, \mathtt{m}, v) \mid cn\ cn \qquad act ::= o \mid \varepsilon$$
$$p ::= \{l \mid s\} \mid \mathtt{idle} \qquad val ::= v \mid \bot$$
$$q ::= \epsilon \mid \{l \mid s\} \mid q\ q \qquad l ::= [\cdots, x \mapsto v, \cdots]$$
$$v ::= o \mid f \mid \ldots$$

Fig. 2 Runtime syntax of `slim ABS`.

- *future binders* $\mathit{fut}(f, \mathit{val})$ where *val*, called *the reply value* may be also $\bot$ meaning that the value has not been computed yet.
- *method invocations* $\mathit{invoc}(o, f, \mathtt{m}, v)$.

The following auxiliary functions are used in the semantic rules (we assume a fixed `slim ABS` program):

- $\mathrm{dom}(l)$ returns the domain of l.
- $l[x \mapsto v]$ is the function such that $(l[x \mapsto v])(x) = v$ and $(l[x \mapsto v])(y) = l(y)$, when $y \neq x$.
- $[\![e]\!]_{(l)}$ returns the value of e by computing the arithmetic expressions and retrieving the value of the identifiers that is stored in l. When e is a future name, the function $[\![\cdot]\!]_{(l)}$ is the identity. Namely $[\![f]\!]_{(l)} = f$.
- `Object.m` returns the term $(T\ x)\{\overline{F\ z}; s\}$ that contains the argument and the body of the method `m`.
- $\mathrm{bind}(o, f, \mathtt{m}, v) = \{[\mathrm{destiny} \mapsto f, x \mapsto v, \overline{z} \mapsto \bot] \mid s[^{o}/_{\mathtt{this}}]\}$, where $\mathtt{Object.m} = (T\ x)\{\overline{F\ z}; s\}$.

The transition relation rules are collected in Figures 3 and 4. They define transitions of objects $\mathit{ob}(o, p, q)$ according to the shape of the statement in p. We focus on rules concerning the concurrent part of `slim ABS`, since the other ones are standard. Rules (Await-True) and (Await-False) model the `await` $e?$ operation: if the (future) value of e has been computed then `await` terminates; otherwise the active process becomes idle. In this case, the object activates one of its queued processes – rule (Activate). Rule (Read-Fut) permits the retrieval of the value returned by a method; the object does not release the execution of the current process until this value has been computed.

The object creation is modeled by (New-Object).

Rule (Async-Call) defines asynchronous method invocation $x = e!\mathtt{m}(e)$. This rule creates a fresh future name that is assigned to the identifier x. The evaluation of the called method is transferred to a different process – see rule (Bind-Mtd). Therefore the caller can progress without waiting for callee's termination. As an example of `slim ABS` semantics, in Figure **??** we have detailed the transitions of the program in Example 2.

```
Int fact_g(Int n){
   Fut<Int> x ;
   Int m ;
   if (n==0) { return 1; }
   else { x = this!fact_g(n-1); m = x.get;
          return n*m; }
}
Int fact_ag(Int n){
   Fut<Int> x ;
   Int m ;
   if (n==0) { return 1; }
   else { x = this!fact_ag(n-1);
          await x?; m = x.get;
          return n*m; }
}
Int fact_nc(Int n){
   Fut<Int> x ;
   Int m ;
   Math z ;
   if (n==0) { return 1 ; }
   else { z = new Object();
          x = z!fact_nc(n-1); m = x.get;
          return n*m; }
}
```

Fig. 5 The class `Math`

1.3 *Samples of concurrent programs in* `slim ABS`

The `slim ABS` code of two concurrent programs are discussed. These codes will be analysed in the following sections.

Example 1 Figure 5 collects three different implementations of the factorial function. The function `fact_g` is the standard definition of factorial: the recursive invocation `this!fact_g(n-1)` is followed by a `get` operation

$$\begin{array}{c}\text{(Skip)}\\ ob(o, \{l \mid \texttt{skip}; s\}, q)\\ \rightarrow ob(o, \{l \mid s\}, q)\end{array}$$

$$\begin{array}{c}\text{(Assign-Local)}\\ \dfrac{x \in \text{dom}(l) \quad v = [\![e]\!]_{(l)}}{\begin{array}{c}ob(o, \{l \mid x = e; s\}, q)\\ \rightarrow ob(o, \{l[x \mapsto v] \mid s\}, q)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(Cond-True)}\\ \dfrac{\texttt{true} = [\![e]\!]_{(l)}}{\begin{array}{c}ob(o, \{l \mid \texttt{if}\ e\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\}; s\}, q)\\ \rightarrow ob(o, \{l \mid s_1; s\}, q)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(Cond-False)}\\ \dfrac{\texttt{false} = [\![e]\!]_{(l)}}{\begin{array}{c}ob(o, \{l \mid \texttt{if}\ e\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\}; s\}, q)\\ \rightarrow ob(o, \{l \mid s_2; s\}, q)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(Await-True)}\\ \dfrac{f = [\![e]\!]_{(l)} \quad v \neq \bot}{\begin{array}{c}ob(o, \{l \mid \texttt{await}\, e\,?; s\}, q)\ fut(f, v)\\ \rightarrow ob(o, \{l \mid s\}, q)\ fut(f, v)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(Await-False)}\\ \dfrac{f = [\![e]\!]_{(l)}}{\begin{array}{c}ob(o, \{l \mid \texttt{await}\, e\,?; s\}, q)\ fut(f, \bot)\\ \rightarrow ob(o, \text{idle}, q \cup \{l \mid \texttt{await}\, e\,?; s\})\ fut(f, \bot)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(Activate)}\\ \dfrac{}{\begin{array}{c}ob(o, \text{idle}, q \cup \{l \mid s\})\\ \rightarrow ob(o, \{l \mid s\}, q)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(Read-Fut)}\\ \dfrac{f = [\![e]\!]_{(l)} \quad v \neq \bot}{\begin{array}{c}ob(o, \{l \mid x = e\texttt{.get}; s\}, q)\ fut(f, v)\\ \rightarrow ob(o, \{l \mid x = v; s\}, q)\ fut(f, v)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(New-Object)}\\ \dfrac{o' = \text{fresh}()}{\begin{array}{c}ob(o, \{l \mid x = \texttt{new Object}; s\}, q)\\ \rightarrow ob(o, \{l \mid x = o'; s\}, q)\\ ob(o'', \text{idle}, \epsilon)\end{array}}\end{array}$$

Fig. 3 Semantics of slim ABS(1).

$$\begin{array}{c}\text{(Async-Call)}\\ \dfrac{o' = [\![e]\!]_{(l)} \quad v = [\![e]\!]_{(l)} \quad f = \text{fresh}(\ )}{\begin{array}{c}ob(o, \{l \mid x = e!\texttt{m}(e); s\}, q)\\ \rightarrow ob(o, \{l \mid x = f; s\}, q)\ invoc(o', f, \texttt{m}, v)\ fut(f, \bot)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(Bind-Mtd)}\\ \dfrac{\{l \mid s\} = \text{bind}(o, f, \texttt{m}, \overline{v}, \text{class}(o))}{\begin{array}{c}ob(o, p, q)\ invoc(o, f, \texttt{m}, \overline{v})\\ \rightarrow ob(o, p, q \cup \{l \mid s\})\end{array}}\end{array}$$

$$\begin{array}{c}\text{(Return)}\\ \dfrac{v = [\![e]\!]_{(l)} \quad f = l(\text{destiny})}{\begin{array}{c}ob(o, \{l \mid \texttt{return}\ e; s\}, q)\ fut(f, \bot)\\ \rightarrow ob(o, \{l \mid s\}, q)\ fut(f, v)\end{array}}\end{array}$$

$$\begin{array}{c}\text{(Context)}\\ \dfrac{cn \rightarrow cn'}{cn\ cn'' \rightarrow cn'\ cn''}\end{array}$$

Fig. 4 Semantics of slim ABS(2).

that retrieves the value returned by the invocation. Yet, `get` does not allow the task to release the cog lock; therefore the task evaluating `this!fact_g(n-1)` is fated to be delayed forever because its object is the same as that of the caller. The function `fact_ag` solves this problem by permitting the caller to release the lock with an explicit `await` operation, before getting the actual value with `x.get`. An alternative solution is defined by the function `fact_nc`, whose code is similar to that of `fact_g`, except for that `fact_nc` invokes `z!fact_nc(n-1)` recursively, where `z` is a new object. This means the task of `z!fact_nc(n-1)` may start without waiting for the termination of the caller.

Programs that are particularly hard to verify are those that may manifest misbehaviours according to the schedulers choices. The following example discusses one case.

Example 2 The Figure 6 defines three methods. Method `m1` asynchronously invokes `m2` on its own argument `y`, passing its own reference as argument . Then it asynchronously invokes `m2` on `this`, passing its same argument `y`. Method `m2` invokes `m3` on the argument `z` and blocks waiting for the result. Method `m3` simply returns.

Next, consider the following main function:

```
{ Object x; Object y; Object z;
  Fut<Int> w ;
  x = new Object;
  y = new Object;
  z = x!m1(y); }
```

```
Int m1(Object y) {
    Fut<Int> h;
    Fut<Int> g ;
    h = y!m2(this);
    g = this!m2(y);
    return 0;
}
Int m2(Object z) {
  Fut<Int> h ;
  Int g;
  h = z!m3();
  g=h.get;
  return 0;
 }
 Int m3(){
   return 0;
}
```

Fig. 6 Methods `m1`, `m2`, and `m3` may or may not incur in a deadlock depending on the scheduler's choice.

The initial configuration is

$$ob(start, \{l \mid s\}, \varnothing)\,cog(\text{start}, start)$$

where $l = [\text{destiny} \mapsto f_{start}, x \mapsto \bot, y \mapsto \bot, z \mapsto \bot]$ and s is the statement of the main function.

ESEMPIO DA AGGIUSTARE

1.4 *Deadlocks*

The definition below identifies deadlocked configurations by detecting chains of dependencies between tasks that cannot progress. To ease the reading, we write

- $p[f.\texttt{get}]$ whenever $p = \{l|s\}$ and s is $x = y.\texttt{get}; s'$ and $[\![y]\!]_{(l)} = f$;
- $p[\texttt{await}\ f]$ whenever $p = \{l|s\}$ and s is $\texttt{await}\ e?; s'$ and $[\![e]\!]_{(l)} = f$;
- $p.f$ whenever $p = \{l|s\}$ and $l(\text{destiny}) = f$.

Definition 1 A configuration cn is *deadlocked* if there are

$$ob(o_0, p_0, q_0), \cdots, ob(o_{n-1}, p_{n-1}, q_{n-1}) \in cn$$
and
$$p'_i \in p_i \cup q_i, \qquad \text{with } 0 \le i \le n-1$$

such that (let $+$ be computed modulo n in the following)

1. $p'_0 = p_0[f_0.\texttt{get}]$ and if $p'_i[f_i.\texttt{get}]$ then $p'_i = p_i$;
2. if $p'_i[f_i.\texttt{get}]$ or $p'_i[\texttt{await}\ f_i]$ then $fut(f_i, \bot) \in cn$ and
   - either $p'_{i+1}[f_{i+1}.\texttt{get}]$ and $p'_{i+1} = \{l_{i+1}|s_{i+1}\}$ and $f_i = l_{i+1}(\text{destiny})$;
   - or $p'_{i+1}[\texttt{await}\ f_{i+1}]$ and $p'_{i+1} = \{l_{i+1}|s_{i+1}\}$ and $f_i = l_{i+1}(\text{destiny})$;
   - or $p'_{i+1} = p_{i+1} = \text{idle}$ and $o_{i+1} = o_{i+2}$ and $p'_{i+2}[f_{i+2}.\texttt{get}]$ (in this case p_{i+1} is idle, by soundness).

A configuration cn is *deadlock-free* if, for every $cn \longrightarrow^* cn'$, cn' is not deadlocked. A `slim` ABS program is *deadlock-free* if its initial configuration is *deadlock-free*.

According to Definition 1, a configuration is deadlocked when there is a circular dependency between processes. The processes involved in such circularities are performing a `get` or `await` synchronisation or they are idle and will never grab the lock because another active process in the same object will not return. We notice that, by Definition 1, at least one active process is blocked on a `get` synchronisation. We also notice that the objects in Definition 1 may be not pairwise different (see example 1 below). The following examples should make the definition clearer; the reader is recommended to instantiate the definition every time.

1. (self deadlock)
$$\begin{array}{l} ob(o_1, \{l_1|x_1 = e_1.\texttt{get}; s_1\}, q_1 \cup \{l_2|s_2\}) \\ fut(f_2, \bot), \end{array}$$
where $[\![e_1]\!]_{(l_1)} = l_2(destiny) = f_2$. In this case, the objects of the Definition 1 are
$$ob(o_1, p_1, q_1) \quad ob(o_1, p_2, q_2 \cup \{l_2|s_2\})$$
where $p'_1 = p_1 = \{l_1|x_1 = e_1.\texttt{get}; s_1\}$, $p'_2 = \{l_2|s_2\}$ and $q_1 = q_2 \cup \{l_2|s_2\}$.
2. (`get`-`await` deadlock)
$$\begin{array}{l} ob(o_1, \{l_1|x_1 = e_1.\texttt{get}; s_1\}, q_1 \cup \{l_3|s_3\}) \\ ob(o_2, \{l_2|\texttt{await}\ e_2?; s_2\}, q_2) \end{array}$$
where $l_3(destiny) = [\![e_2]\!]_{l_2}$, $l_2(destiny) = [\![e_1]\!]_{l_1}$. In this case, o_2 cannot progress because it is waiting for a result of a process that cannot be scheduled (because it in the queue of o_1).
3. (`get`-idle deadlock)
$$\begin{array}{l} ob(o_1, \{l_1|x_1 = e_1.\texttt{get}; s_1\}, q_1) \\ ob(o_3, \{l_3|x_3 = e_3.\texttt{get}; s_3\}, q_3 \cup \{l_2|s_2\}) \\ ob(o_5, \{l_5|x_5 = e_5.\texttt{get}; s_5\}, q_5 \cup \{l_4|s_4\}) \\ fut(f_1, \bot),\ fut(f_2, \bot),\ fut(f_4, \bot) \end{array}$$
where $f_2 = l_2(destiny) = [\![e_1]\!]_{l_1}$, $f_4 = l_4(destiny) = [\![e_3]\!]_{l_3}$, $f_1 = l_1(destiny) = [\![e_5]\!]_{l_5}$.

A deadlocked configuration has at least one object that is stuck (the one performing the `get` instruction). This means that the configuration may progress, but future configurations will still have one object stuck.

Proposition 1 *If cn is deadlocked and $cn \longrightarrow cn'$ then cn' is deadlocked as well.*

Definition 1 is about runtime entities that have no static counterpart. Therefore we consider a notion weaker than deadlocked configuration. This last notion will be used in the Appendices to demonstrate the correctness of the inference system in Section 2.

Definition 2 A configuration cn has

(i) a *dependency* (o, o') if
$$ob(o, \{l|x = e.\texttt{get}; s\}, q), ob(o', p', q') \in cn$$
with $[\![e]\!]_{(l)} = f$ and
  (a) either $fut(f, \bot) \in cn$, $l'(\text{destiny}) = f$ and $\{l'|s'\} \in p' \cup q'$;
  (b) or $invoc(o', f, \texttt{m}, v) \in cn$.

(ii) a *dependency* $(o, o')^{\texttt{w}}$ if
$$ob(o, p, q), ob(o', p', q') \in cn$$
and $\{l|\texttt{await}\ e?; s\} \in p \cup q$ and $[\![e]\!]_{(l)} = f$ and
  (a) either $fut(f, \bot) \in cn$, $l'(\text{destiny}) = f$ and $\{l'|s'\} \in p' \cup q'$;
  (b) or $invoc(o', f, \texttt{m}, v) \in cn$.

Given a set A of dependencies, let the get-*closure* of A, noted $A^{\texttt{get}}$, be the least set such that

1. $A \subseteq A^{\texttt{get}}$;
2. if $(o, o') \in A^{\texttt{get}}$ and $(o', o'')^{[\texttt{w}]} \in A^{\texttt{get}}$ then $(o, o'') \in A^{\texttt{get}}$, where $(o', o'')^{[\texttt{w}]}$ denotes either the pair (o', o'') or the pair $(o', o'')^{\texttt{w}}$.

A configuration contains a *circularity* if the get-closure of its set of dependencies has a pair (o, o).

Proposition 2 *If a configuration is deadlocked then it has a circularity. The converse is false.*

Proof The statement is a straightforward consequence of the definition of deadlocked configuration. To show that the converse is false, consider the configuration

$$\begin{array}{l} ob(o_1, \{l_1 | x_1 = e_1.\texttt{get}; s_1\}, q_1) \\ ob(o_2, \{l_3 | \texttt{return } e_3\}, q_3 \cup \{l_2 | \texttt{await } e_2?; s_2\}) \quad cn \end{array}$$

where $l_3(destiny) = [\![e_1]\!]_{l_1}$, $l_1(destiny) = [\![e_2]\!]_{l_2}$,. This configuration has the dependencies

$$\{(o_1, o_2), (o_2, o_1)^{\texttt{w}}\}$$

whose get-closure contains the circularity (o_1, o_1). However the configuration is not deadlocked. □

ESEMPIO DA AGGIUSTARE

2 Contracts and the contract inference system

The deadlock detection framework we present in this paper relies on abstract descriptions, called *contracts*, that are extracted from programs by an inference system. The syntax of these descriptions, which is defined in Figure 7, uses *record names* $X, Y, Z, \cdots$, and *future names* $f, f', \cdots$.

Future records $\mathbb{r}$, which encode the values of expressions in contracts, may be one of the following:

– a dummy value $\_$ that models primitive types,
– a record name X that represents a place-holder for a value and can be instantiated by substitutions,
– an object name o,
– and $o \rightsquigarrow \mathbb{r}$, which specifies that accessing $\mathbb{r}$ requires control of the object o (and that the control is to be released once the method has been evaluated). The future record $o \rightsquigarrow \mathbb{r}$ is associated with method invocations: o is the object on which the method is invoked. The name o in $o \rightsquigarrow \mathbb{r}$ will be called *root* of the future record.

Contracts $\mathbb{c}$ collect the method invocations and the dependencies inside statements. In addition to 0, $f.(o, o')$, and $f.(o, o')^{\texttt{w}}$ that respectively represent the empty behaviour, the dependencies due to a get and an await operation, we have the contract that deal with method invocations: $\nu f.\texttt{m}(\mathbb{r}, \mathbb{r}') \to \mathbb{s}$ specifies that a new future f is associated to the invocation of method m on an object $\mathbb{r}$, with argument $\mathbb{r}'$, and an object $\mathbb{s}$ will be returned. The composite contracts $\mathbb{c} \fatsemi \mathbb{c}'$ and $\mathbb{c} + \mathbb{c}'$ define the abstract behaviour of sequential compositions and conditionals, respectively.

Example 3 As an example of contracts, let us discuss the terms:

(a) $\nu f.\texttt{m}(o_1) \to o_1' \fatsemi$
$\nu f'.\texttt{m}(o_2) \to o_2'$;

(b) $\nu f.\texttt{m}(o_1) \to o_1' \fatsemi f.(o, o_1)$
$\nu f'.\texttt{m}(o_2) \to o_2' \fatsemi f'.(o, o_2)^{\texttt{w}}$;

The contract (a) defines a sequence of two invocations of method m. We notice that the names o_1' and o_2' are free: this indicates that m returns a new object. As we will see below, a slim ABS expression with this contract is `x!m() ; y!m() ;`.

The contract (b) defines an invocation of m followed by a get statement and an invocation followed by an await. The object o is the caller. A slim ABS expression retaining this contract is `u = x!m() ; w = u.get ; v = y!m() ; await v? ;`.

The inference of contracts uses two additional syntactic categories: $\mathbb{x}$ of future record values and $\mathbb{z}$ of typing values. The former one extends future records with *future names*, which are used to carry out the *alias analysis*. In particular, every local variable of methods and every parameter of future type is associated to a future name. Assignments between these terms, such as $x = y$, amounts to copying future names instead of the corresponding values (x and y become aliases). The category $\mathbb{z}$ collects the typing values of future names, which are either $\mathbb{r}$, for *unsynchronised futures*, or $\mathbb{r}^{\checkmark}$, for *synchronised ones* (see the comments below).

The abstract behaviour of methods is defined by *method contracts* $\mathbb{r}(\mathbb{s})\ \{\mathbb{c}\}\ \mathbb{r}'$, where $\mathbb{r}$ is the future record of the receiver of the method, $\mathbb{s}$ is the future record of the argument, $\mathbb{c}$ is the abstract behaviour of the body, and $\mathbb{r}'$ is the future record of the returned object.

The subterm $(\mathbb{r}, \mathbb{r}')$ of the method contract is called *header*; $\mathbb{s}$ is called *returned future record*. We assume that object and record names in the header occur linearly. Object and record names in the header *bind* the object and record names in $\mathbb{c}$ and in $\mathbb{s}$. The header

$\mathbb{r} ::= \_ \mid X \mid o \mid o \rightsquigarrow \mathbb{r}$	future record
$\mathbb{c} ::= 0 \mid f.(o,o') \mid f.(o,o')^{\mathtt{w}} \mid \nu f.\mathtt{m}(\mathbb{r},\mathbb{r}') \to \mathbb{s} \mid \mathbb{c} \fatsemi \mathbb{c} \mid \mathbb{c} + \mathbb{c} \mid \mathbb{c} \parallel \mathbb{c}$	contract
$\mathbb{x} ::= \mathbb{r} \mid f$	extended future record
$\mathbb{z} ::= \mathbb{r} \mid \mathbb{r}^{\checkmark}$	future reference values

Fig. 7 Syntax of future records and contracts.

and the returned future record, written $(\mathbb{r},\mathbb{r}') \to \mathbb{s}$, are called *contract signature.* In a method contract $(\mathbb{r},\mathbb{r}')\ \{\mathbb{c}\}\ \mathbb{s}$, object and record names occurring in $\mathbb{c}$ or $\mathbb{c}'$ or $\mathbb{s}$ may be *not bound* by header. These *free names* correspond to `new` instructions and will be replaced by fresh object names during the analysis.

2.1 *Inference of contracts*

Contracts are extracted from `slim ABS` programs by means of an inference algorithm. Figures 8 and 10 illustrate the set of rules. The following auxiliary operator is used:

– $mname(\overline{M})$ returns the sequence of method names in the sequence $\overline{M}$ of method declarations.

The inference algorithm uses constraints $\mathcal{U}$, which are defined by the following syntax

$$\mathcal{U} ::= \mathtt{true} \mid c = c' \mid \mathbb{r} = \mathbb{r}' \mid (\mathbb{r},\mathbb{r}') \to \mathbb{r}'' \preceq (\mathbb{s},\mathbb{s}') \to \mathbb{s}'' \\ \mid \mathcal{U} \wedge \mathcal{U}$$

where $\mathtt{true}$ is the constraint that is always true; $\mathbb{r} = \mathbb{r}'$ is a classic unification constraint between terms; $(\mathbb{r},\mathbb{r}') \to \mathbb{r}'' \preceq (\mathbb{s},\mathbb{s}') \to \mathbb{s}''$ is a *semi-unification* constraint; the constraint $\mathcal{U} \wedge \mathcal{U}'$ is the conjunction of $\mathcal{U}$ and $\mathcal{U}'$. We use *semi-unification* constraints [?] to deal with method invocations: basically, in $(\mathbb{r},\mathbb{r}') \to \mathbb{r}'' \preceq (\mathbb{s},\mathbb{s}') \to \mathbb{s}''$, the left hand side of the constraint corresponds to the method's formal parameter, $\mathbb{r}$ being the record of `this`, $\mathbb{r}'$ being the records of the parameter and $\mathbb{r}''$ being the record of the returned value, while the right hand side corresponds to the actual parameters of the call, and the actual returned value. The meaning of this constraint is that the actual parameters and returned value must match the specification given by the formal parameters, like in a standard unification: the necessity of semi-unification appears when we call several times the same method. Indeed, there, unification would require that the actual parameters of the different calls must all have the same records, while with semi-unification all method calls are managed independently.

The judgments of the inference algorithm have a typing context Γ mapping variables to extended future records, future names to future name values and methods to their signatures. They have the following form:

– $\Gamma \vdash_o e : \mathbb{x}$ for pure expressions e and $\Gamma \vdash_o f : \mathbb{z}$ for future names f, where o is the name of the object executing the expression and $\mathbb{x}$ and $\mathbb{z}$ are their inferred values.
– $\Gamma \vdash_o z : \mathbb{r}, \mathbb{c} \rhd \mathcal{U} \mid \Gamma'$ for expressions with side effects z, where o, and $\mathbb{x}$ are as for pure expressions e, $\mathbb{c}$ is the contract for z created by the inference rules, $\mathcal{U}$ is the generated constraint, and Γ' is the environment Γ *with updates* of variables and future names. We use the same judgment for pure expressions; in this case $\mathbb{c} = 0$, $\mathcal{U} = \mathtt{true}$ and $\Gamma' = \Gamma$.
– for statements s: $\Gamma \vdash_o s : \mathbb{c} \rhd \mathcal{U} \mid \Gamma'$ where o, $\mathbb{c}$ and $\mathcal{U}$ are as before, and Γ' is the environment obtained after the execution of the statement. The environment may change because of variable updates.

Since Γ is a function, we use the standard predicates $x \in \mathrm{dom}(\Gamma)$ or $x \notin \mathrm{dom}(\Gamma)$. Moreover, given a function Γ, we define $\Gamma[x \mapsto \mathbb{x}]$ to be the following function

$$\Gamma[x \mapsto \mathbb{x}](y) = \begin{cases} \mathbb{x} & \text{if } y = x \\ \Gamma(y) & \text{otherwise} \end{cases}$$

We also let $\Gamma|_{\{x_1,\cdots,x_n\}}$ be the function

$$\Gamma|_{\{x_1,\cdots,x_n\}}(y) = \begin{cases} \Gamma(y) & \text{if } y \in \{x_1,\cdots,x_n\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Moreover, provided that $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \varnothing$, the environment $\Gamma + \Gamma'$ be defined as follows

$$(\Gamma + \Gamma')(x) \stackrel{def}{=} \begin{cases} \Gamma(x) & \text{if } x \in \mathrm{dom}(\Gamma) \\ \Gamma'(x) & \text{if } x \in \mathrm{dom}(\Gamma') \end{cases}$$

Finally,we let

$$\mathtt{Fut}(\Gamma) \stackrel{def}{=} \{x \mid \Gamma(x) \text{ is a future name}\}.$$

The inference rules for expressions and future names are reported in Figure 8. They are straightforward, except for (T-Value) that performs the dereference of variables and return the future record stored in the future name of the variable. (T-Pure) lifts the judgment of a expressions and addresses

(T-Var)
$$\frac{\Gamma(x) = \mathbb{x}}{\Gamma \vdash_o x : \mathbb{x}}$$

(T-Fut)
$$\frac{\Gamma(f) = \mathbb{z}}{\Gamma \vdash_o f : \mathbb{z}}$$

(T-Value)
$$\frac{\Gamma \vdash_o e : f \qquad \Gamma \vdash_o f : (\mathbb{r}, \mathbb{c})^{[\checkmark]}}{\Gamma \vdash_o e : \mathbb{r}}$$

(T-Val)
$$\frac{e \quad \textit{primitive value or arithmetic-and-bool-exp}}{\Gamma \vdash_o e : \_}$$

(T-Pure)
$$\frac{\Gamma \vdash_o e : \mathbb{r}}{\Gamma \vdash_o e : \mathbb{r}, 0 \rhd \texttt{true} \mid \Gamma}$$

expressions with side effects

(T-Get)
$$\frac{\Gamma \vdash_o x : f \qquad \Gamma \vdash_o f : \mathbb{r} \qquad X, o' \text{ fresh} \qquad \Gamma' = \Gamma[f \mapsto \mathbb{r}^{\checkmark}]}{\Gamma \vdash_o x.\texttt{get} : X, f.(o, o') \rhd \mathbb{r} = o' \rightsquigarrow X \mid \Gamma'}$$

(T-Get-tick)
$$\frac{\Gamma \vdash_o x : f \qquad \Gamma \vdash_o f : \mathbb{r}^{\checkmark} \qquad X, o' \text{ fresh}}{\Gamma \vdash_o x.\texttt{get} : X, 0 \rhd \mathbb{r} = o' \rightsquigarrow X \mid \Gamma}$$

(T-New)
$$\frac{o' \text{ fresh}}{\Gamma \vdash_o \texttt{new Object} : o', 0 \rhd \texttt{true} \mid \Gamma}$$

(T-AInvk)
$$\frac{\Gamma \vdash_o e : \mathbb{r} \qquad \Gamma \vdash_o e' : \mathbb{s} \qquad X, o', f \text{ fresh}}{\Gamma \vdash_o e!\texttt{m}(e') : f, 0 \rhd o' = \mathbb{r} \wedge \texttt{m} \preceq (\mathbb{r}, \mathbb{s}) \to X \mid \Gamma[f \mapsto (o' \rightsquigarrow X, \nu f.\texttt{m}(\mathbb{r}, \mathbb{s}) \to X)]}$$

Fig. 8 Contract inference for expressions and expressions with side effects.

statements

T-Skip
$$\Gamma \vdash_o \texttt{skip} : 0 \rhd \texttt{true} \mid \Gamma$$

(T-Assign)
$$\frac{\Gamma(x) = \mathbb{x} \qquad \Gamma \vdash_o z : \mathbb{x}', \mathbb{c} \rhd \mathcal{U} \mid \Gamma'}{\Gamma \vdash_o x = z : \mathbb{c} \rhd \mathcal{U} \mid \Gamma'[x \mapsto \mathbb{x}']}$$

(T-Await)
$$\frac{\Gamma \vdash_o e : f \qquad \Gamma \vdash_o f : \mathbb{r} \qquad X, o' \text{ fresh} \qquad \Gamma' = \Gamma[f \mapsto \mathbb{r}^{\checkmark}]}{\Gamma \vdash_o \texttt{await } e? : f.(o, o')^{\mathtt{w}} \rhd \mathbb{r} = o' \rightsquigarrow X \mid \Gamma'}$$

(T-Await-Tick)
$$\frac{\Gamma \vdash_o e : f \qquad \Gamma \vdash_o f : \mathbb{r}^{\checkmark} \qquad X, o' \text{ fresh}}{\Gamma \vdash_o \texttt{await } e? : 0 \rhd \mathbb{r} = o' \rightsquigarrow X \mid \Gamma}$$

(T-If)
$$\frac{\begin{array}{c}\Gamma \vdash_o e : \texttt{Bool} \qquad \Gamma \vdash_o s_1 : \mathbb{c}_1 \rhd \mathcal{U}_1 \mid \Gamma_1 \\ \Gamma \vdash_o s_2 : \mathbb{c}_2 \rhd \mathcal{U}_2 \mid \Gamma_2 \qquad \mathcal{U} = \Big(\bigwedge_{x \in \mathrm{dom}(\Gamma)} \Gamma_1(x) = \Gamma_2(x) \Big) \wedge \Big(\bigwedge_{x \in \texttt{Fut}(\Gamma)} \Gamma_1(\Gamma_1(x)) = \Gamma_2(\Gamma_2(x)) \Big) \qquad \Gamma' = \Gamma_1 + \Gamma_2|_{\{f \mid f \notin \Gamma_2(\texttt{Fut}(\Gamma))\}}\end{array}}{\Gamma \vdash_o \texttt{if } e \,\{\, s_1 \,\}\, \texttt{else} \,\{\, s_2 \,\} : \mathbb{c}_1 + \mathbb{c}_2 \rhd \mathcal{U}_1 \wedge \mathcal{U}_2 \wedge \mathcal{U} \mid \Gamma'}$$

(T-Seq)
$$\frac{\Gamma \vdash_o s_1 : \mathbb{c}_1 \rhd \mathcal{U}_1 \mid \Gamma_1 \qquad \Gamma_1 \vdash_o s_2 : \mathbb{c}_2 \rhd \mathcal{U}_2 \mid \Gamma_2}{\Gamma \vdash_o s_1; s_2 : \mathbb{c}_1 \fatsemi \mathbb{c}_2 \rhd \mathcal{U}_1 \wedge \mathcal{U}_2 \mid \Gamma_2}$$

(T-Return)
$$\frac{\Gamma \vdash_o e : \mathbb{r} \qquad \Gamma(\texttt{destiny}) = \mathbb{r}'}{\Gamma \vdash_o \texttt{return } e : 0 \rhd \mathbb{r} = \mathbb{r}' \mid \Gamma}$$

Fig. 9 Contract inference for statements.

pure expression to a judgment similar to those for expressions with side-effects. This expedient allows us to simplify rules for statements.

Figure 8 also reports inference rules for expressions with side effects. Rule (T-Get) deals with the x.`get` synchronisation primitive and returns the contract $f.(o, o')$, where (o, o') represents a dependency between the cog of the object executing the expression and the root of the expression. We also observe that the rule updates the environment by check-marking the value of the future name of x. This allows subsequent `get` (and `await`) operations on the same future name not to modify the contract (in fact, in this case they are operationally equivalent to the `skip` statement) – see (T-Get-Tick).

Rule (T-New) returns a record with a new object name. Rule (T-AInvk) derives contracts for asynchronous invocations. Since the dependencies created by these invocations influence the dependencies of the synchronised contract only if a subsequent `get` or `await` operation is performed, the rule stores the invocation into a fresh future name of the environment and returns the contract 0. This models `slim ABS` semantics that lets asynchronous invocations be synchronised by explicitly getting or awaiting on the corresponding future variable, see rules (T-Get) and (T-Await). The future name storing the invocation is returned by the judgment.

The inference rules for statements are collected in Figure 9. The first rule define the inference of contracts for assignment(rule (T-Assign)).

Rule (T-Await) and (T-AwaitTick) deal with the `await` synchronisation when applied to a simple future lookup x?. They are similar to the rules (T-Get) and (T-GetTick).

Rule (T-If) defines contracts for conditionals. In this case we collect the contracts $\mathbb{c}_1$ and $\mathbb{c}_2$ of the two branches, with the intended meaning that the dependencies defined by $\mathbb{c}_1$ and $\mathbb{c}_2$ are always kept separated. As regards the environments, the rule constraints the two environments Γ_1 and Γ_2 produced by typing of the two branches to *be the same* on variables in $\mathrm{dom}(\Gamma)$ *and* on the values of future names bound to variables in $\mathtt{Fut}(\Gamma)$. However, the two branches may have different unsynchronised invocations that are not bound to any variable. The environment $\Gamma_1 + \Gamma_2|_{\{f \mid f \notin \Gamma_2(\mathtt{Fut}(\Gamma))\}}$ allows us to collect all them.

Rule (T-Seq) defines the sequential composition of contracts. Rule (Return) constrains the record of `destiny`, which is an identifier introduced by (T-Method), shown in Figure 10, for storing the return record.

The rules for method and class declarations are defined in Figure 10. Rule (T-Method) derives the method contract of T `m` $(\overline{T\ x})\{\overline{F\ u}; s\}$ by typing s in an environment extended with `this`, `destiny` (that will be set by `return` statements, see (T-Return)), the arguments $\overline{x}$, and the local variables $\overline{u}$. In order to deal with alias analysis of future variables, we separate fields, parameters, arguments and local variables with future types from the other ones. In particular, we associate future names to the former ones and bind future names to record variables.

The rule (T-Program) yields an *abstract class table* that associates a method contract with every method name, and it derives the contract of a `slim` ABS program by typing the main function in the same way as it was a body of a method.

The contract class tables of the classes in a program derived by the rule (Class), will be noted CCT. We will address the contract of `m` by CCT(`m`). In the following, we assume that every `slim` ABS program is a triple $(\mathrm{CT}, \{\overline{T\ x\ ;\ s}\}, \mathrm{CCT})$, where CT is the class table, $\{\overline{T\ x\ ;\ s}\}$ is the main function, and CCT is its contract class table. By rule (Program), analysing (the deadlock freedom of) a program, amounts to verifying the contract of the main function with a record for `this` that has associated a special object name *start*.

ESEMPIO da aggiustare

We notice that the inference system of contracts discussed in this section is modular because, when programs are organised in different modules, it partially supports the separate contract inference of modules with a well-founded ordering relation (for example, if there are two modules, classes in the second module use definitions or methods in the first one, but not conversely). In this case, if a module `B` includes a module `A` then a patch to a class of `B` amounts to inferring contracts for `B` only. On the contrary, a patch to a class of `A` may also require a new contract inference of `B`.

2.2 Correctness results

In our system, the ill-typed programs are those manifesting a failure of the semiunification process, which does not address misbehaviours. In particular, a program may be well-typed and still manifest a deadlock. In fact, in systems with *behavioural types*, one usually demonstrates that

1. in a well-typed program, every configuration cn has a behavioural type, let us call it $\mathtt{bt}(cn)$;
2. if $cn \to cn'$ then there is a relationship between $\mathtt{bt}(cn)$ and $\mathtt{bt}(cn')$;
3. the relationship in 2 preserves a given property (in our case, deadlock-freedom).

Item 1, in the context of the inference system of this section, means that the program has a contract class table. Its proof needs a contract system for configurations, which we have defined in Appendix **??**. The theorem corresponding to this item is Theorem **??**.

Item 2 requires the definition of a relation between contracts, called *later stage relation* in Appendix **??**. This later stage relation is a syntactic relationship between contracts whose basic law is that a method invocation is larger than the instantiation of its method contract (the other laws, except $0 \trianglelefteq \mathbb{c}$ and $\mathbb{c}_i \trianglelefteq \mathbb{c}_1 + \mathbb{c}_2$, are congruence laws).

The statement that relates the later stage relationship to `slim` ABS reduction is Theorem **??**. It is worth to observe that all the theoretical development up-to this point are useless if the later stage relation conveyed no relevant property. This is the purpose of item 3, which requires the definition of *contract models* and the proof that deadlock-freedom is preserved by the models of contracts in later stage relation. The reader can find the proofs of these statements in the Appendices **??** and **??** (they correspond to the two analysis techniques that we study).

(T-Method)
$$\frac{o, X, X_r, \overline{Y}, \overline{Z}, \overline{f}, Z \textit{ fresh} \qquad \Gamma + \texttt{this} : o + x{:}X + \overline{y{:}Y} + \overline{z{:}f} + \overline{f{:}Z} + \texttt{destiny} : X_r \vdash_o s : \mathbb{c} \rhd \mathcal{U} \mid \Gamma''}{\Gamma \vdash T\ \texttt{m}\ (T_x\ x)\{\overline{T_y\ y;}\ \overline{\texttt{Fut<}T_z\texttt{>}\ z;}\ s\}\ :\quad (o, X)\{\mathbb{c}\}\ X_r \rhd \mathcal{U} \wedge (o, X) \rightarrow X_r = \texttt{m}}$$

(T-Program)
$$\frac{\Gamma \vdash \overline{M} : \overline{\mathbb{C}} \rhd \overline{\mathcal{U}} \qquad \overline{X}, \overline{X'}, \overline{f} \textit{ fresh} \qquad \Gamma + \overline{x{:}X} + \overline{x'{:}f} + \overline{f{:}(X', 0)} \vdash_{start} s : \mathbb{c} \rhd \mathcal{U} \mid \Gamma'}{\Gamma \vdash \overline{M}\ \{\overline{T\ x;}\ \overline{\texttt{Fut<}T'\texttt{>}\ x';}\ s\} : \overline{mname(M) \mapsto \mathbb{C}}, \mathbb{c} \rhd \mathcal{U} \wedge \overline{\mathcal{U}}}$$

Fig. 10 Contract rules of method and class declarations and programs.